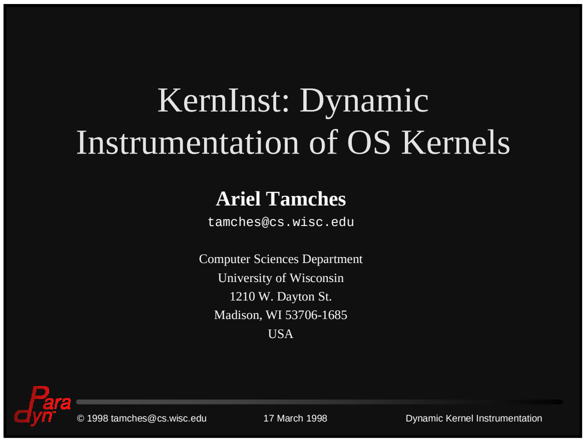
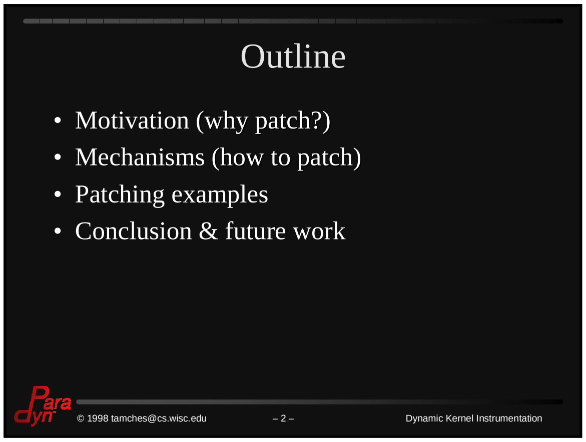


KernInst: Dynamic Instrumentation of OS Kernels

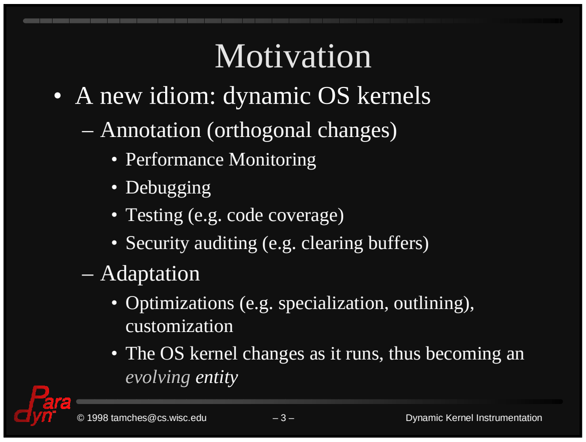
Ariel Tamches
`tamches@cs.wisc.edu`

Computer Sciences Department
University of Wisconsin
1210 W. Dayton St.
Madison, WI 53706-1685
USA



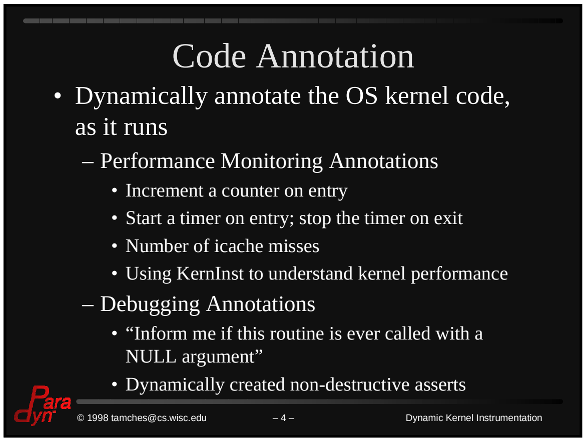
Outline

- Motivation (why patch?)
- Mechanisms (how to patch)
- Patching examples
- Conclusion & future work



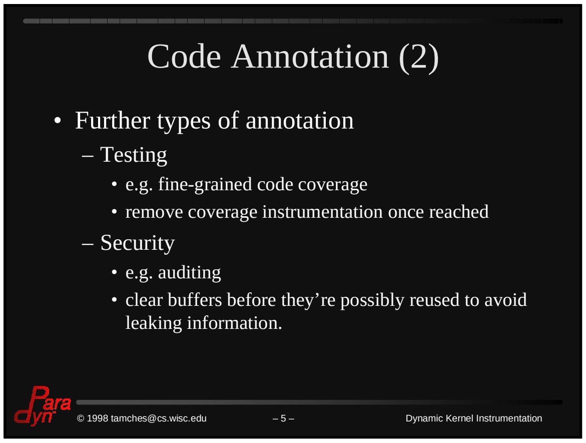
Motivation

- A new idiom: dynamic OS kernels
  - Annotation (orthogonal changes)
    - Performance Monitoring
    - Debugging
    - Testing (e.g. code coverage)
    - Security auditing (e.g. clearing buffers)
  - Adaptation
    - Optimizations (e.g. specialization, outlining), customization
    - The OS kernel changes as it runs, thus becoming an *evolving entity*



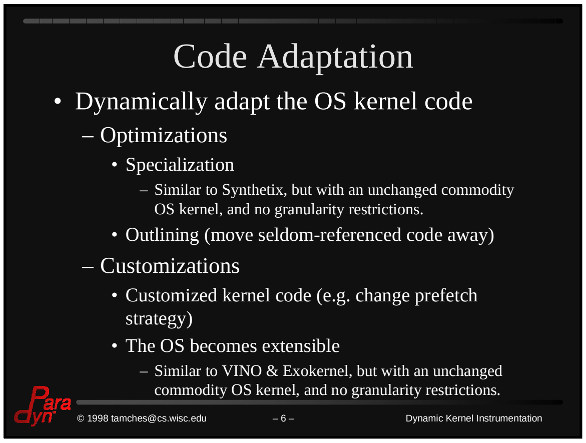
Code Annotation

- Dynamically annotate the OS kernel code, as it runs
  - Performance Monitoring Annotations
    - Increment a counter on entry
    - Start a timer on entry; stop the timer on exit
    - Number of icache misses
    - Using KernInst to understand kernel performance
  - Debugging Annotations
    - “Inform me if this routine is ever called with a NULL argument”
    - Dynamically created non-destructive asserts



Code Annotation (2)

- Further types of annotation
  - Testing
    - e.g. fine-grained code coverage
    - remove coverage instrumentation once reached
  - Security
    - e.g. auditing
    - clear buffers before they’re possibly reused to avoid leaking information.



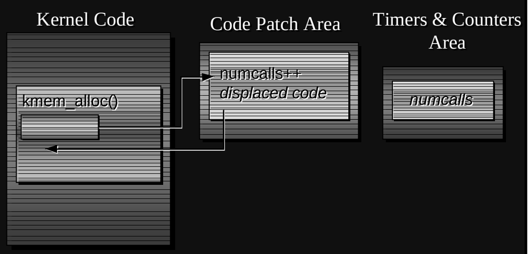
Code Adaptation

- Dynamically adapt the OS kernel code
  - Optimizations
    - Specialization
      - Similar to Synthetix, but with an unchanged commodity OS kernel, and no granularity restrictions.
    - Outlining (move seldom-referenced code away)
  - Customizations
    - Customized kernel code (e.g. change prefetch strategy)
    - The OS becomes extensible
      - Similar to VINO & Exokernel, but with an unchanged commodity OS kernel, and no granularity restrictions.

Features

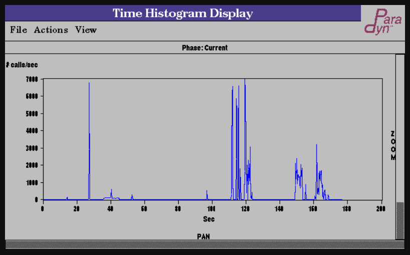
- On-the-fly, dynamic: while kernel is running
- On stock, production (COTS) OS kernels
  - We use Solaris 2.5 running on UltraSPARC.
- Fine-grained instrumentation: granularity of instructions or basic blocks, not just entire functions.
- General purpose: augment OS kernel with arbitrary code sequences.

Simple Example of Patching


Kernel Code
Code Patch Area
Timers & Counters Area
kmem_alloc()
numcalls++
displaced code
numcalls

- We've annotated kmem_alloc: "Increment a counter on entry"

Simple Example of Patching


Time Histogram Display
File
Actions
View
Phase: Current
# calls/sec
7000
6000
5000
4000
3000
2000
1000
0
0
20
40
60
80
100
120
140
160
180
200
Sec
ZOOM
PAN

Mechanisms: How to Patch

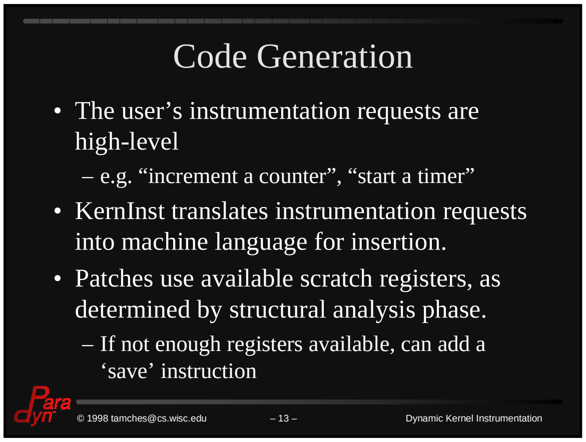
- Bootstrapping
- Structural Analysis
- Code Generation
- Code Splicing
- Data Collection

Bootstrapping

- How can KernInst (a user-level process)
  - Allocate necessary in-kernel code and data space for instrumentation?
  - Get the kernel's symbol information?
  - Mechanism for modifying the kernel's address space (/dev/kmem insufficient)
- Solution: create a driver (/dev/kerninst)
  - Installed into kernel on-the-fly (common OS feature)
  - Communicate with KernInst via file operations.

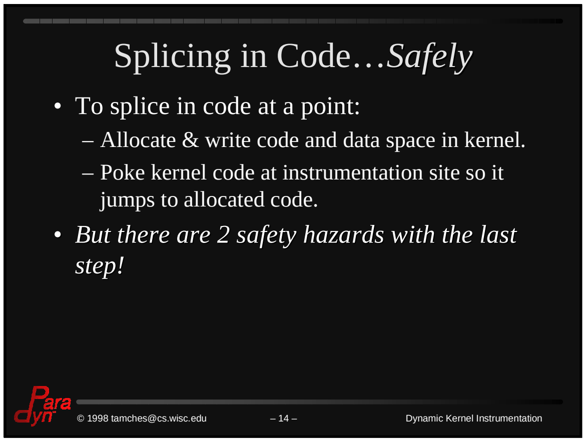
Structural Analysis

- Need free registers at instrumentation points
  - For jumping to dynamically inserted code.
  - For executing dynamically inserted code.
- Run live register analysis algorithm
  - Get function addresses from symbol table
  - Parse kernel's in-memory machine code to find basic blocks.
- Interprocedural Analysis
  - Possible since all code is present in memory



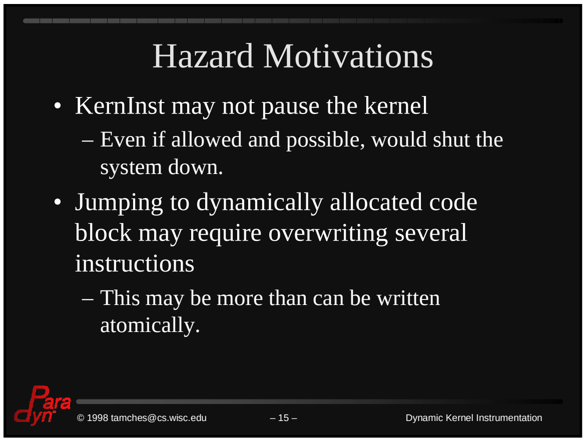
Code Generation

- The user’s instrumentation requests are high-level
  - e.g. “increment a counter”, “start a timer”
- KernInst translates instrumentation requests into machine language for insertion.
- Patches use available scratch registers, as determined by structural analysis phase.
  - If not enough registers available, can add a ‘save’ instruction



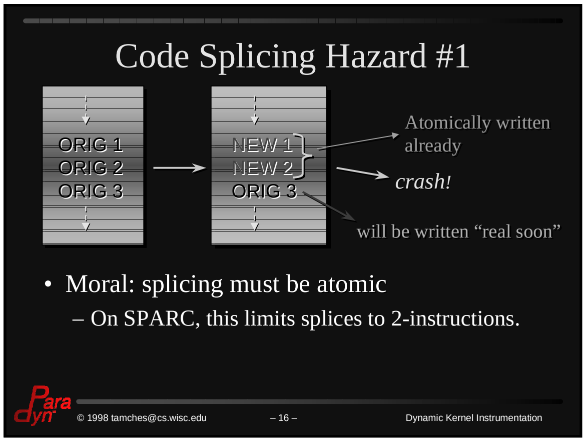
Splicing in Code…*Safely*

- To splice in code at a point:
  - Allocate & write code and data space in kernel.
  - Poke kernel code at instrumentation site so it jumps to allocated code.
- *But there are 2 safety hazards with the last step!*



Hazard Motivations

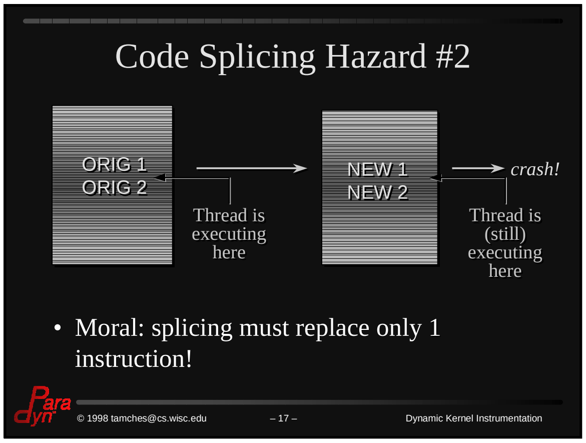
- KernInst may not pause the kernel
  - Even if allowed and possible, would shut the system down.
- Jumping to dynamically allocated code block may require overwriting several instructions
  - This may be more than can be written atomically.



Code Splicing Hazard #1

ORIG 1
ORIG 2
ORIG 3
NEW 1
NEW 2
ORIG 3
Atomically written already
crash!
will be written “real soon”

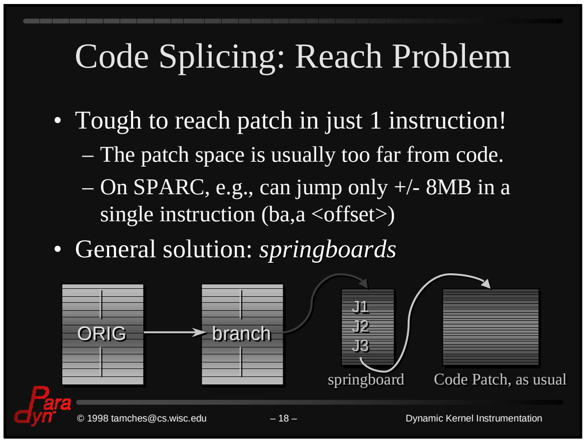
- Moral: splicing must be atomic
  - On SPARC, this limits splices to 2-instructions.



Code Splicing Hazard #2

ORIG 1
ORIG 2
Thread is executing here
NEW 1
NEW 2
crash!
Thread is (still) executing here

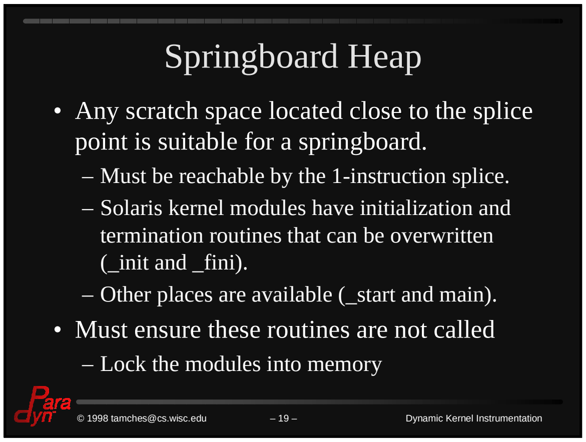
- Moral: splicing must replace only 1 instruction!



Code Splicing: Reach Problem

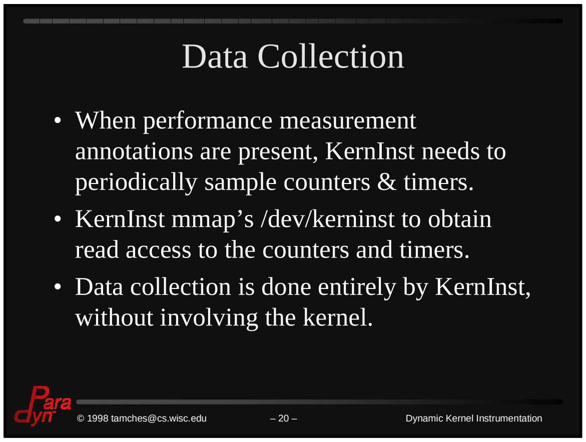
- Tough to reach patch in just 1 instruction!
  - The patch space is usually too far from code.
  - On SPARC, e.g., can jump only +/- 8MB in a single instruction (ba,a <offset>)
- General solution: *springboards*

ORIG
branch
J1
J2
J3
springboard
Code Patch, as usual



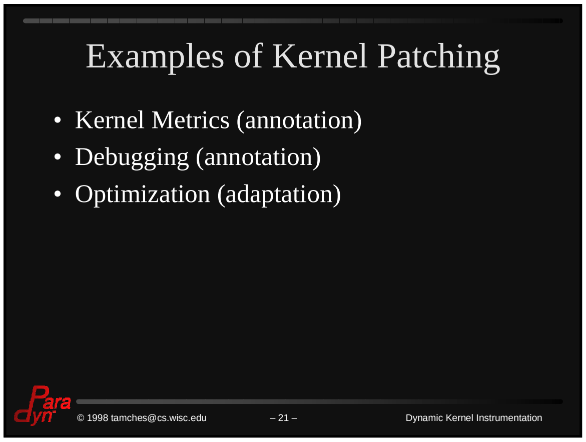
Springboard Heap

- Any scratch space located close to the splice point is suitable for a springboard.
  - Must be reachable by the 1-instruction splice.
  - Solaris kernel modules have initialization and termination routines that can be overwritten (_init and _fini).
  - Other places are available (_start and main).
- Must ensure these routines are not called
  - Lock the modules into memory



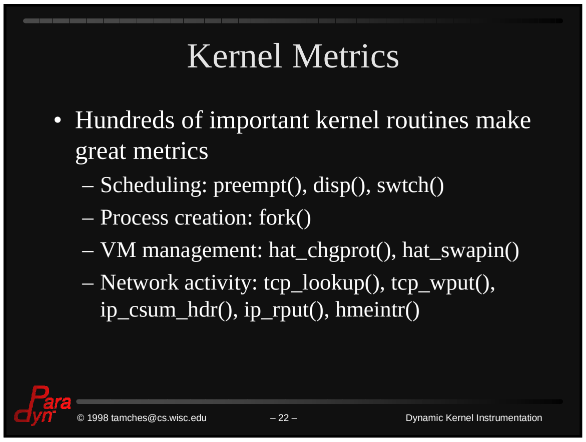
Data Collection

- When performance measurement annotations are present, KernInst needs to periodically sample counters & timers.
- KernInst mmap's /dev/kerninst to obtain read access to the counters and timers.
- Data collection is done entirely by KernInst, without involving the kernel.



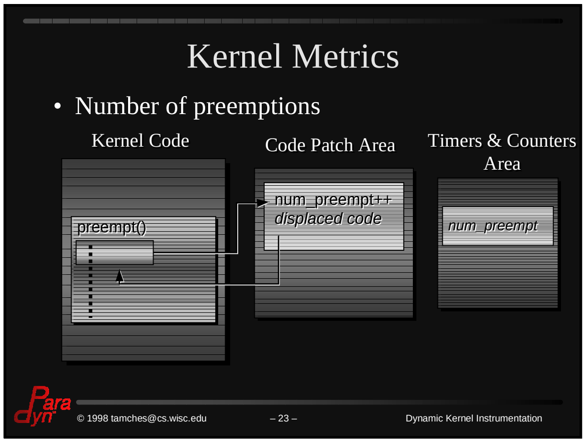
Examples of Kernel Patching

- Kernel Metrics (annotation)
- Debugging (annotation)
- Optimization (adaptation)



Kernel Metrics

- Hundreds of important kernel routines make great metrics
  - Scheduling: preempt(), disp(), swtch()
  - Process creation: fork()
  - VM management: hat_chgprot(), hat_swapin()
  - Network activity: tcp_lookup(), tcp_wput(), ip_csum_hdr(), ip_rput(), hmeintr()



Kernel Metrics

- Number of preemptions

Kernel Code

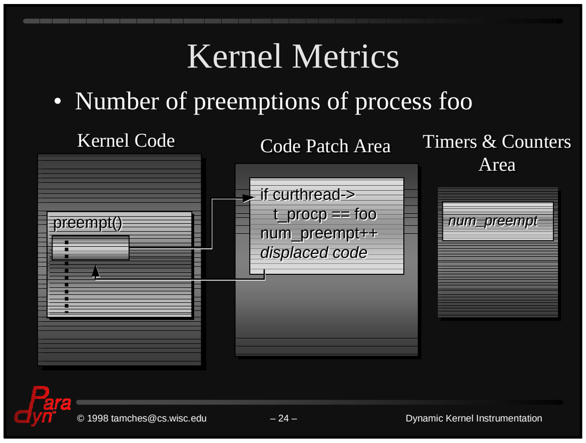
preempt()

Code Patch Area

num_preempt++
displaced code

Timers & Counters Area

num_preempt



Kernel Metrics

- Number of preemptions of process foo

Kernel Code

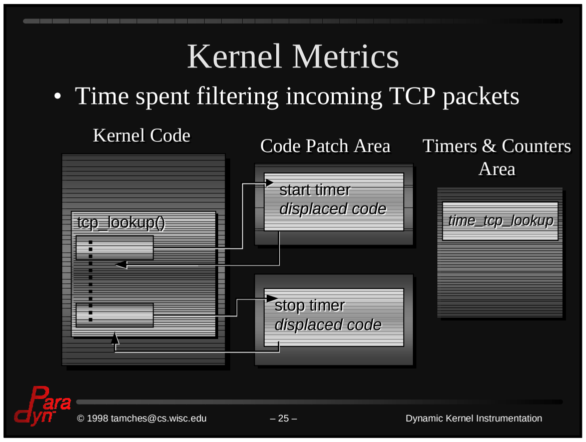
preempt()

Code Patch Area

if curthread->
t_procp == foo
num_preempt++
displaced code

Timers & Counters Area

num_preempt



Kernel Metrics

- Time spent filtering incoming TCP packets

Kernel Code

Code Patch Area

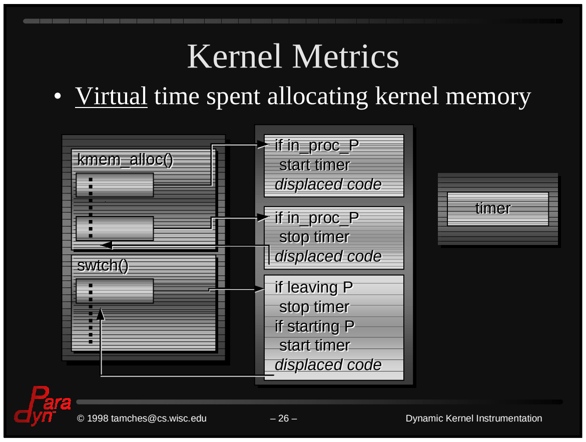
Timers & Counters Area

tcp_lookup()

start timer
displaced code

stop timer
displaced code

time_tcp_lookup



Kernel Metrics

- Virtual time spent allocating kernel memory

kmem_alloc()

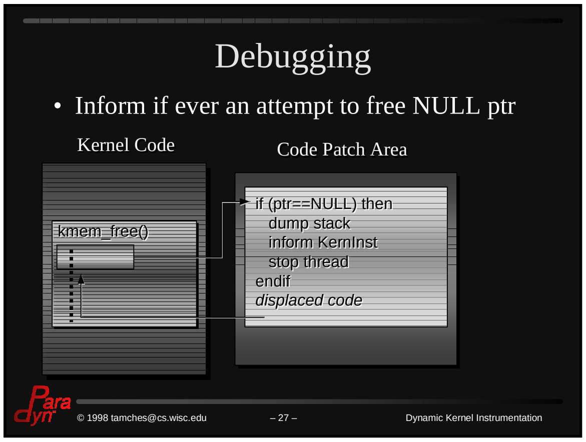
switch()

if in_proc_P
start timer
displaced code

if in_proc_P
stop timer
displaced code

if leaving P
stop timer
if starting P
start timer
displaced code

timer



Debugging

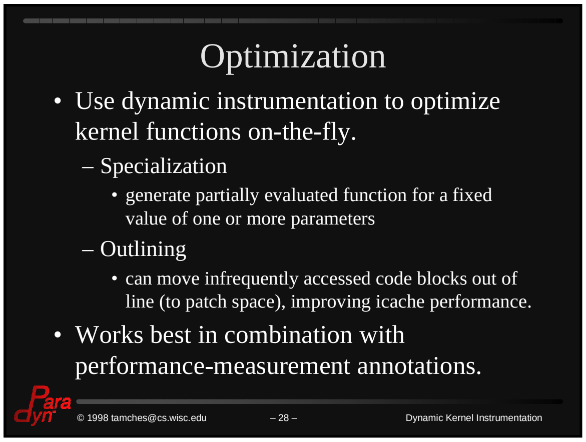
- Inform if ever an attempt to free NULL ptr

Kernel Code

Code Patch Area

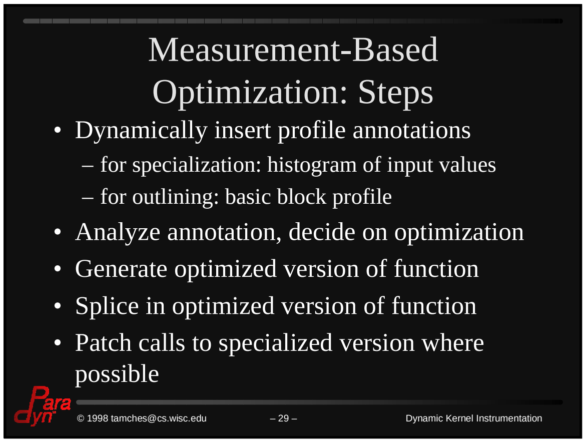
kmem_free()

```
if (ptr==NULL) then
   dump stack
   inform KernInst
   stop thread
endif
displaced code
```



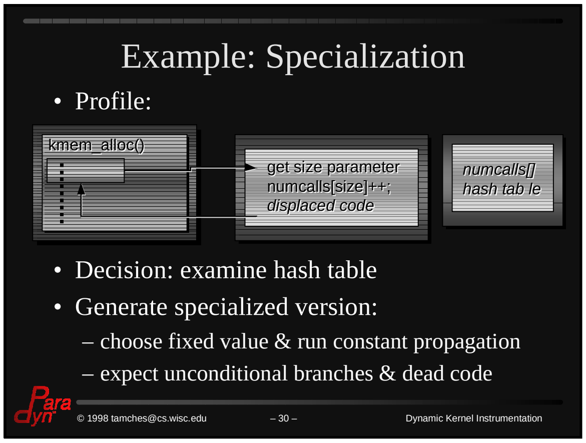
Optimization

- Use dynamic instrumentation to optimize kernel functions on-the-fly.
  - Specialization
    - generate partially evaluated function for a fixed value of one or more parameters
  - Outlining
    - can move infrequently accessed code blocks out of line (to patch space), improving icache performance.
- Works best in combination with performance-measurement annotations.



Measurement-Based Optimization: Steps

- Dynamically insert profile annotations
  - for specialization: histogram of input values
  - for outlining: basic block profile
- Analyze annotation, decide on optimization
- Generate optimized version of function
- Splice in optimized version of function
- Patch calls to specialized version where possible



Example: Specialization

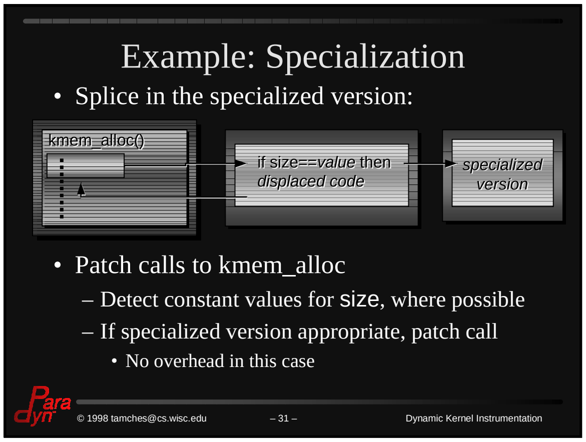
- Profile:

kmem_alloc()

get size parameter
numcalls[size]++;
displaced code

numcalls[]
hash table

- Decision: examine hash table
- Generate specialized version:
  - choose fixed value & run constant propagation
  - expect unconditional branches & dead code



Example: Specialization

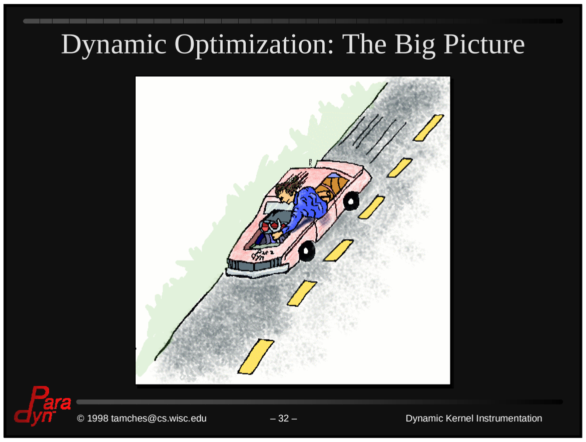
- Splice in the specialized version:

kmem_alloc()

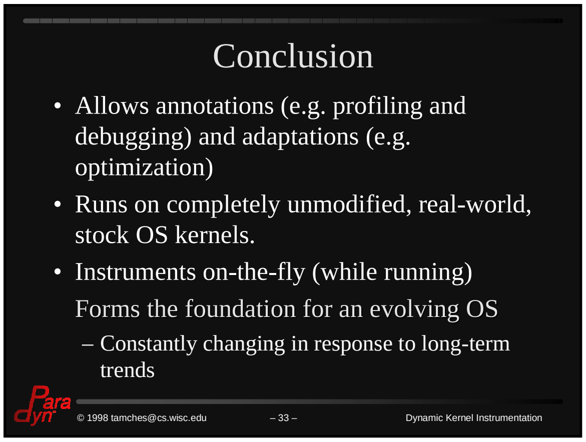
if size==*value* then
displaced code

specialized version

- Patch calls to kmem_alloc
  - – Detect constant values for size, where possible
  - – If specialized version appropriate, patch call
    - No overhead in this case



Dynamic Optimization: The Big Picture



Conclusion

- Allows annotations (e.g. profiling and debugging) and adaptations (e.g. optimization)
- Runs on completely unmodified, real-world, stock OS kernels.
- Instruments on-the-fly (while running)

Forms the foundation for an evolving OS

- – Constantly changing in response to long-term trends